ACube

Charte d'architecture

Version 1.1 du 20/02/2010

Etat : Validé

SUIVI DES MODIFICATIONS

Version	Rédaction	Description	Vérification	Date
1.0	S. Péguet	Initialisation		22/03/07
1.1	JP. Wilsch	Ajout Lise v3		18/02/10

LISTE DE DIFFUSION

Organisation	Nom	Info	Commentaire	Validation
		☐	☐	☐

SOMMAIRE

FIGURES

1 INTRODUCTION

L'architecture technique multi-niveaux ACube repose sur les standards Internet (W3C/J2EE).

L'objectif de ce document n'a pas pour vocation de décrire dans le détail l'ensemble des fonctionnalités techniques mises en oeuvre dans cette architecture mais à lister les principes techniques fondateurs de cette architecture pour préciser sa modularité et les compétences nécessaires au développement de l'applicatif attendu.

2 PRINCIPES TECHNIQUES FONDATEURS ACUBE

2.1 XML : FORMAT D'ECHANGE DE L'INFORMATION

Le standard XML comme format d'échange de l'information intervient dans les différentes logiques de programmation au sein de l'architecture multi-niveaux ACube :

- **Logique multi-canal :** le serveur a pour vocation, par l'intermédiaire du standard XML, de fournir des flux métiers indépendants du canal d'accès.
- **Logique multi-format :** le standard XML permet de séparer systématiquement la logique de contenu informatif de la logique du format d'affichage ou de rendu.
- **Logique multi-langue :** l'architecture modulaire du serveur offre la possibilité de cloisonner l'ensemble des libellés affichés sous la forme de paramétrage directement monté en mémoire par l'intermédiaire de fichier XML. Ce cloisonnement permet ainsi de gérer plusieurs langues pour un même métier.
- **Logique de configuration :** l'ensemble de la configuration technique propre à un projet fonctionnel est effectué de préférence par l'intermédiaire d'un paramétrage XML.
- **Logique de communication inter-applicative ou avec un partenaire :** l'ensemble des flux informatifs liés à un processus métier dont les fonctionnalités sont partagées sur plusieurs applicatifs ou plusieurs systèmes d'information (dialogue partenaire) doit de préférence être formaté en XML.

Pour éviter tout problème de traitement, il est conseillé d'effectuer une vérification par DTD (Document Type Definition) ou XSD (Schéma XML) pour certifier que les flux XML (XSD) ou la configuration XML (DTD) sont « valides et bien formés ». Cette recommandation est surtout utile lors de l'implémentation de la logique de communication inter-applicative ou avec un partenaire.

2.2 PRINCIPE DU CLIENT RICHE

2.2.1 ENRICHISSEMENT DU CLIENT

Indépendamment du type d'ergonomie retenu, les sites web modernes nécessitent la prise en charge d'un certain nombre de traitements par la couche client :

- **Gestion de la navigation** en prenant en compte les problématiques suivantes :
  - Construction des urls en relatif.
  - Appel d'une url donnée en fonction de l'option sélectionnée à partir d'une boîte de sélection.
  - Gestion de l'historique de la navigation (BACK, FORWARD et REFRESH).
- **Gestion des formulaires** en prenant en compte les problématiques suivantes :
  - Validation d'un formulaire par la touche ENTER.
  - Eviter les appels multiples de validation d'un formulaire lors de plusieurs clicks.
  - Désactivation des actions utilisateur en attente de l'affichage du résultat de la validation d'un formulaire.
  - Contrôles de surface pour valider le formalisme de la saisie utilisateur.
  - Affranchissement des limitations liées aux champs de saisie Internet pour s'approcher de ceux liés au modèle client/serveur (onglet, arborescence, assistant à la saisie, attachement/détachement, calendrier pour la saisie de date, data grid, tableur…)

- **Gestion des effets dynamiques** :
  - Changement d'images.
  - Affichage/Désaffichage d'une zone dynamique.
  - Modification du contenu d'une zone dynamique en fonction d'une action utilisateur.
  - Affichage d'une information dans la barre de statut.
  - Modification du style d'un objet du document.
  - Drag'n'drop.
- **Gestion des fenêtres** :
  - Gestion du multi-fenêtrage.
  - Création et fermeture d'une fenêtre fille.
  - Impression du contenu d'une fenêtre.
- **Gestion de contenu** par le client :
  - Génération d'une partie du contenu d'un document en fonction d'une action utilisateur ou en fonction d'une charte graphique pour simplifier la maintenance de cette charte.
  - Génération du contenu d'une fenêtre surgissante.
- **Gestion du contexte** client par l'intermédiaire de cookie session.

L'ensemble de ces enrichissements fait l'objet d'une norme de développement pour expliciter leur mise en œuvre facilitée par l'utilisation d'un framework.

2.2.2 PRISE EN COMPTE D'UNE ERGONOMIE COMPLEXE PROPRE A UNE APPLICATION DE GESTION

Contrairement à une ergonomie dite « classique » consistant à baser le client sur une navigation par page basée sur des liens hypertextes ou sur validations de formulaire, une ergonomie complexe basée sur les principes du **client riche** est appliquée de préférence.

Ainsi, une seule page HTML statique est nécessaire pour gérer un métier ou un cas d'utilisation. L'ensemble des résultats liés aux actions client est affiché ou désaffiché par le biais de zones dynamiques (tag <DIV>). Les données résultantes de ces actions sont appelées et montées en mémoire au format XML et converties au format XHTML par le navigateur.

Par cet intermédiaire, seuls les flux de contenu informatif (données du SI) sont générés par le serveur. Ceci permet pour la gestion d'un métier de prendre en compte la persistance des données sur le navigateur dans le cycle de vie des actions de l'utilisateur.

Autre point positif, l'accès en mémoire sous forme d'arbre XML aux données rapatriées offre la possibilité d'interagir avec celles-ci. Ainsi, le navigateur prend en charge certaines règles techniques comme le tri d'une liste, l'ajout d'un élément après confirmation du serveur dans l'arbre XML pour réafficher la liste des éléments…

Ce type d'ergonomie a donc pour avantages :

- une plus grande capacité de montée en charge du fait d'une sollicitation moindre des couches multi-canal et services du serveur.
- une souplesse ergonomique plus importante par l'utilisation de zones dynamiques.
- des flux d'échange de taille plus faible.
- une utilisation optimum du cache navigateur par l'utilisation plus importante d'objets statiques (.html et .js) et la gestion de la persistance de données des flux XML dynamiques.

L'inconvénient principal de ce type d'ergonomie consiste en l'augmentation de la complexité de programmation des IHMs. La prise en compte de flux XML pour générer par le navigateur le code XHTML associé en JavaScript est un concept novateur dont ACube a été précurseur avant d'opter pour la technologie AJAX.

D'où la nécessité d'utiliser un **framework ergonomique client riche** pour pouvoir masquer la complexité de leurs mises en oeuvre et ainsi diminuer les coûts de maintenance et de développement (voir §4.1 pour plus de détail).

2.2.3 GARANT DE LA CHARTE GRAPHIQUE ET ERGONOMIQUE

L'autre atout majeur du principe du client riche appliqué dans le cadre d'ACube est d'associer à la gestion technique de l'enrichissement du client et de l'ergonomie complexe, des APIs de génération d'éléments constitutifs d'une page en accord avec la charte graphique et ergonomique.

Ainsi, l'utilisation du framework JavaScript ergonomique client riche ACube garantit une cohérence systématique graphique et ergonomique propre à une application de gestion ACube. La maintenance du site vis à vis de la charte graphique s'avère ainsi simplifiée car elle n'impacte que le framework et non l'ensemble des pages du site.

2.3 LES PARADIGMES DE PROGRAMMATION SERVEUR J2EE (DESIGN PATTERNS)

L'architecture applicative technique J2EE ACube repose sur plusieurs design patterns (ou paradigme de programmation) reconnus permettant une meilleure maintenance des applications et une flexibilité vers de nouvelles technologies.

2.3.1 LE PARADIGME DE PROGRAMMATION DAO

Le paradigme de programmation **« Data Access Object »** ou DAO permet de résoudre la problématique de séparation de la logique métier et de la logique rapatriement de données. Ainsi, l'accès à plusieurs accesseurs (exemple d'accesseur : JDBC, WebServices, JNDI, EJB, JavaMail, LDAP…) pour un même objet métier est entièrement transparent, permettant ainsi de s'affranchir du support de données et de s'ouvrir sur toute évolution possible.

L'accès à ces différents accesseurs s'effectue à travers des APIs qui sont spécifiques à l'accesseur. L'implémentation change donc en fonction de l'accesseur pour un même objet métier.

L'utilisation du design pattern DAO permet de faire abstraction de l'accesseur et de l'encapsuler au sein d'une interface indépendante de la méthode d'accès aux données. L'accesseur gère alors la connexion et la communication avec la source de données.

Dans cette structure, l'objet métier est décomposé en :

- un « Value Object » ou VO qui représente un objet « serializable » manipulant les données issues des différentes sources. Ce dernier est créé au sein d'un DAO
- un « Objet Delegate » qui encapsule les accesseurs au travers d'une interface standard (DAO) qui est déclinée en fonction des méthodes d'accès (DAO JDBC, DAO LDAP, …)

Le DAO représente l'interface aux méthodes statiques qui manipulent les VO. Il est implémenté par les différents accesseurs spécifiques (JNDI, EJB, JDBC…) nécessaires à l'objet.

Le DAO permet de cacher la complexité d'accès aux données tandis que le delegate permet de « typer » cet accès en sélectionnant le bon accesseur.

La structure simplifiée de ce design pattern est représentée ci-dessous :

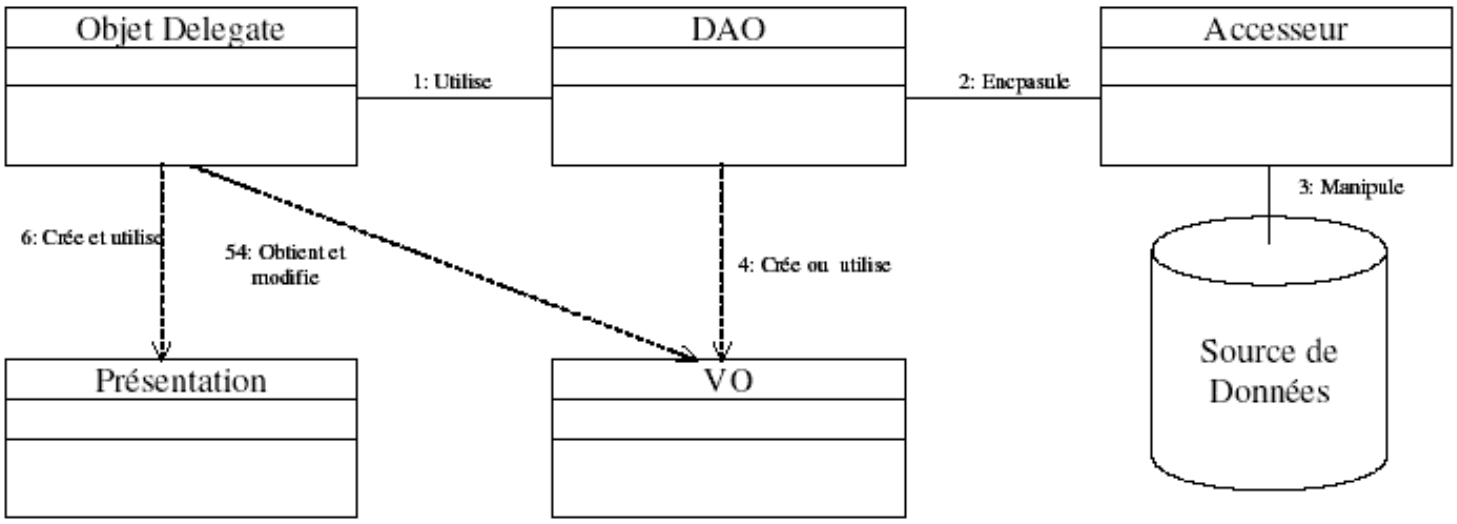

Figure 1 : Design pattern DAO

Ce paradigme conseillé par Sun fournit une architecture ouverte vers de futures évolutions et favorise la mutualisation du code nécessaire aux accesseurs propres aux besoins du projet.

2.3.2 LE PARADIGME DE PROGRAMMATION MVC

Les concepteurs d'applications Internet/Intranet s'accordent aujourd'hui sur un modèle applicatif sur lequel doit être construite toute application Internet robuste, évolutive et maintenable. Ce modèle en trois couches prend en charge les interactions avec les utilisateurs, les traitements métiers de l'application et l'accès aux sources de données auxquelles est connectée l'application. Ces couches sont généralement complétées par un socle technique en charge de services transversaux partagés (journalisation, configuration, …).

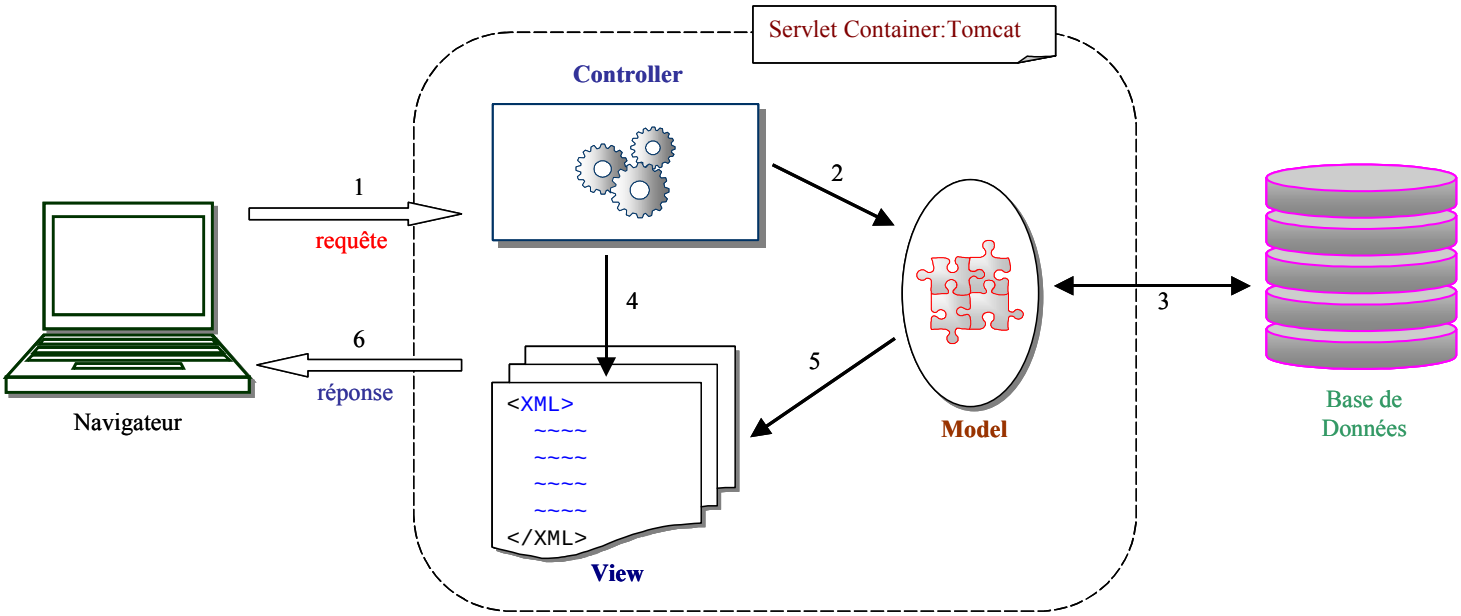

Figure 2 : Design pattern MVC

Le contrôleur : une servlet unique directement invoquée récupère les informations provenant de l'utilisateur et contrôle la validité des actions demandées par le client. Le contrôleur appelle le bon traitement applicatif dans le modèle (voir ci-après), et renvoie vers la vue appropriée (voir ci-après). C'est donc à ce niveau que la logique de navigation et la vérification de la validité de la session utilisateur (timeout…) est mutualisée.

Le modèle : couche principale en charge des traitements applicatifs et des objets métiers. Le modèle ne doit pas manipuler d'éléments liés à l'interface graphique ou faire référence à des servlets ou des vues.

Les objets modélisant le métier proviennent directement de l'analyse fonctionnelle. L'intérêt de ces objets est de cacher la complexité de l'accès aux données et d'offrir une interface simple et proche du domaine que l'on désire représenter.

La vue : génération des formats de sortie attendue pour affichage résultant de l'action de l'utilisateur. Ce code devra être aussi court que possible (les traitements se déroulent dans le modèle et les mécanismes d'accès aux données sont encapsulés dans les objets métiers).

Dans ce modèle, le cycle de vie d'une requête est alors le suivant :

- Le client envoie une requête à l'application. La requête est prise en charge par le contrôleur (étape 1).
- Le contrôleur analyse la requête et réoriente celle-ci de manière à exécuter les traitements nécessaires à la satisfaction de la requête. Il sollicite pour cela les objets métiers (étape 2).
- Les objets métiers fournissent des données (gestion des requêtes SQL) au contrôleur (étape 3).
- Le contrôleur encapsule les données métiers dans des JavaBeans, sélectionne la vue qui sera en charge de la construction de la réponse et lui transmet les JavaBeans contenant les données métier (étape 4).
- La vue construit la réponse (étape 5) en faisant appel aux JavaBeans qui lui ont été transmis et l'envoie au navigateur (étape 6).
- Lorsque nécessaire, pour le traitement d'erreurs par exemple, le contrôleur traite directement la requête, sélectionne la vue de sortie et lui transmet les informations (étape 4) dont elle a besoin afin d'afficher un message approprié (étape 6).

Ce modèle en couches présente de nombreux avantages parmi lesquels :

- il facilite le découpage du travail et les responsabilités des développeurs en définissant plusieurs profils types d'intervenants,
- il améliore la réutilisation et la mutualisation du code dans une optique de qualité des développements,
- il facilite la maintenance. En cas de bogue ou de problème de performance, ce découpage permet de détecter plus rapidement la source du problème et de corriger la partie défaillante sans toucher aux autres éléments,
- il a été totalement implémenté à travers des Framework disponibles gratuitement.

2.4 LA TRAÇABILITE

2.4.1 UTILITE DE LA TRAÇABILITE

La gestion des logs est un enjeu majeur de la réalisation de tout projet fonctionnel sous cette architecture. Effectivement, ceux-ci permettent de résoudre et de corriger un grand nombre de problèmes, aussi bien lors de la phase de développement, que dans la phase de maintenance de l'application ou durant le fonctionnement de l'applicatif.

Les données contenues dans ces logs sont une source d'informations très importante pour de nombreux acteurs (Chef de projet, maîtrise d'ouvrage, exploitant, auditeur de sécurité, support technique…)

2.4.2 METHODOLOGIE

Quel que soit l'outil utilisé pour enregistrer des messages de logs, le point fondamental pour l'exploitabilité d'un applicatif est la pertinence du libellé des messages des traces. Un message d'erreur doit permettre un diagnostic précis de la raison du dysfonctionnement par une personne qui ne connaît pas le code source. Le libellé doit donc être clair et contenir l'ensemble des informations nécessaires à la compréhension de ce qui s'est réellement passé.

Exemple :

Si une application échoue dans l'ouverture de son fichier de configuration, le message ne doit pas être du style "erreur d'ouverture du fichier de configuration" mais "erreur d'ouverture du fichier de configuration <le nom complet du fichier> (<libellé de l'erreur système x>)" en ayant eu le soin de préciser le code et le type de l'événement tracé.

Enfin, le code retour de toute fonction et appel système doit être testé, et en cas d'échec l'application doit envoyer à l'utilisateur un message correspondant à l'erreur.

L'utilisation d'un code événement systématique et standardisé pour tous les logs est très importante. Elle permet de créer un manuel de suivi de l'application qui peut être distribué au support technique et au service chargé de l'administration. Ce dernier permet de faire la relation entre les erreurs relayées et la marche à suivre lorsqu'on les rencontre. Cette procédure permet d'optimiser les démarches et de gagner un temps précieux lors d'un problème grave pour détecter l'origine exacte de l'erreur.

Il ne faut jamais oublier que même les cas les plus improbables ou jamais constatés sur les plates-formes de développement peuvent se produire en production. Il est donc fondamental de tout tester afin d'éviter la propagation d'erreurs et d'alerter l'opérateur d'administration avec le diagnostic de la **première** erreur (traçabilité système).

Il existe plusieurs niveaux de traçabilité qui se complètent les uns les autres :

- La **traçabilité applicative**, elle permet de suivre le cheminement d'un utilisateur dans l'applicatif, de lister précisément les opérations réalisées par l'application, de déboguer l'application en développement, etc…
- La **traçabilité système**, elle permet d'effectuer un suivi et une surveillance des erreurs et des dysfonctionnements par le pilotage métier.
- La **traçabilité des outils** utilisés, elle permet de compléter les traces précédentes en utilisant la traçabilité intégrée dans les outils propres à l'architecture.

Remarque :

La traçabilité applicative fait partie intégrante du développement des applicatifs. Elle doit être intégrée dans le code des composants par l'équipe de développement.

2.4.2.1 FORMAT GENERAL DES LOGS

Le format général des logs lié à la traçabilité applicative est le suivant :

<date>|<applicatif>|<site>|<type>|<code_evenement>|<utilisateur>|<commentaires>

Date : correspond à la date aaaa:MM:jj:hh:mm:ss de l'évènement tracé

Applicatif : code de l'applicatif (chaîne de 5 caractères)

Site : code du site

Type : type de l'événement tracé (ERROR, SECURITE, DEBUG, CRITIQUE ou INFO)

Code évènement : code de l'évènement tracé (code sur 4 chiffres)

Utilisateur : identifiant de l'utilisateur

Commentaires : description de l'événement tracé

3 ARCHITECTURE TECHNIQUE ACUBE

3.1 ARCHITECTURE RESEAU GLOBALE

L'architecture ACube peut reposer sur différentes architectures réseaux selon les besoins. En voici une liste non exhaustive :

- Architecture Intranet Centralisée ayant pour principe de regrouper les serveurs ACube sur un site unique avec un point d'accès réseau LAN, MAN ou WAN.
- Architecture Intranet Décentralisée permettant de fournir une solution de contournement pour les postes client disposant d'une liaison LAN, MAN ou WAN trop faible ou accédant à des applications nécessitant des échanges de flux importants ou volumineux. Le principe technique de proxy/reverse proxy est alors déployé pour décentraliser tout ou partie des fonctionnalités métier embarqués dans l'applicatif déployé.
- Architecture Interne ayant pour principe de regrouper les serveurs ACube sur un site unique avec un point d'accès réseau Internet.
- Architecture Extranet ayant pour principe d'ouvrir les serveurs présents sur l'Architecture Intranet Centralisée sur le point d'accès réseau Internet.
- Architecture Partenaire ayant principe de regrouper les serveurs ACube sur un site unique avec plusieurs points d'accès réseaux aux différents partenaires (LS, Ader…).

L'ensemble de ces architectures réseaux sont contrôlées et démilitarisées par une politique de sécurité et de DMZ.

3.2 ARCHITECTURE GLOBALE DU SERVEUR J2EE

3.2.1 FRAMEWORK J2EE LISE V 2.X

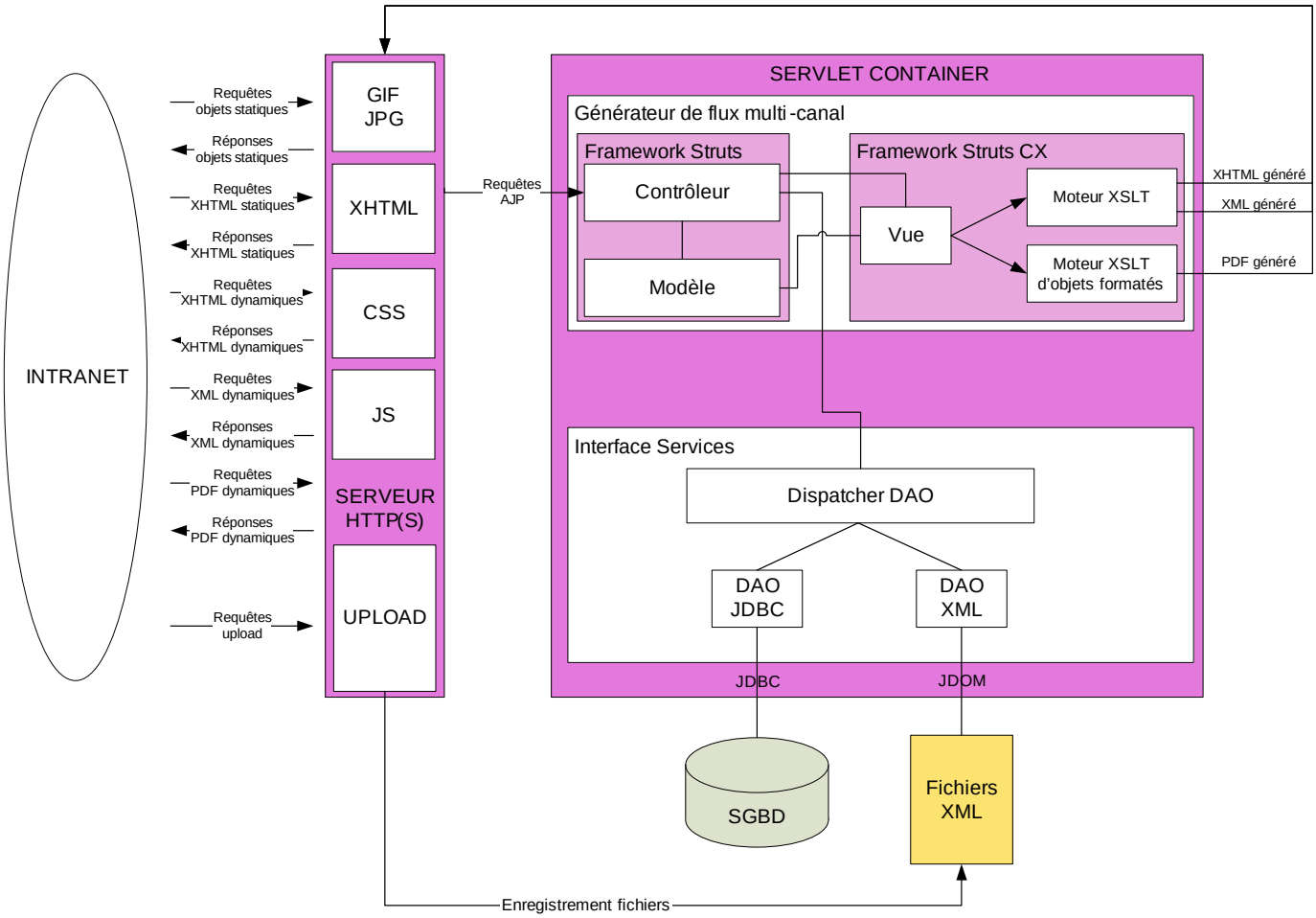

Figure 3 : Architecture globale du serveur ACube (Lise 2.x)

Cette figure illustre les principales briques **Open Source** de l'architecture **multi-niveaux** ACube (Lise 2.x), soient :

- Apache pour le serveur Web.
- Tomcat pour le « Servlet Container ».
- Connecteur AJP entre Apache Web Server et Tomcat.
- JRE 1.5.0 comme moteur d'exécution Java.
- Struts 1.x pour la navigation
- Struts CX pour la génération des flux générés
- Xalan pour le moteur XSLT.
- Apache FOP pour le moteur XSLT d'objet formaté.
- SQL Server , Oracle ou MySql pour la base de données relationnelle.

3.2.2 FRAMEWORK J2EE LISE V 3.X

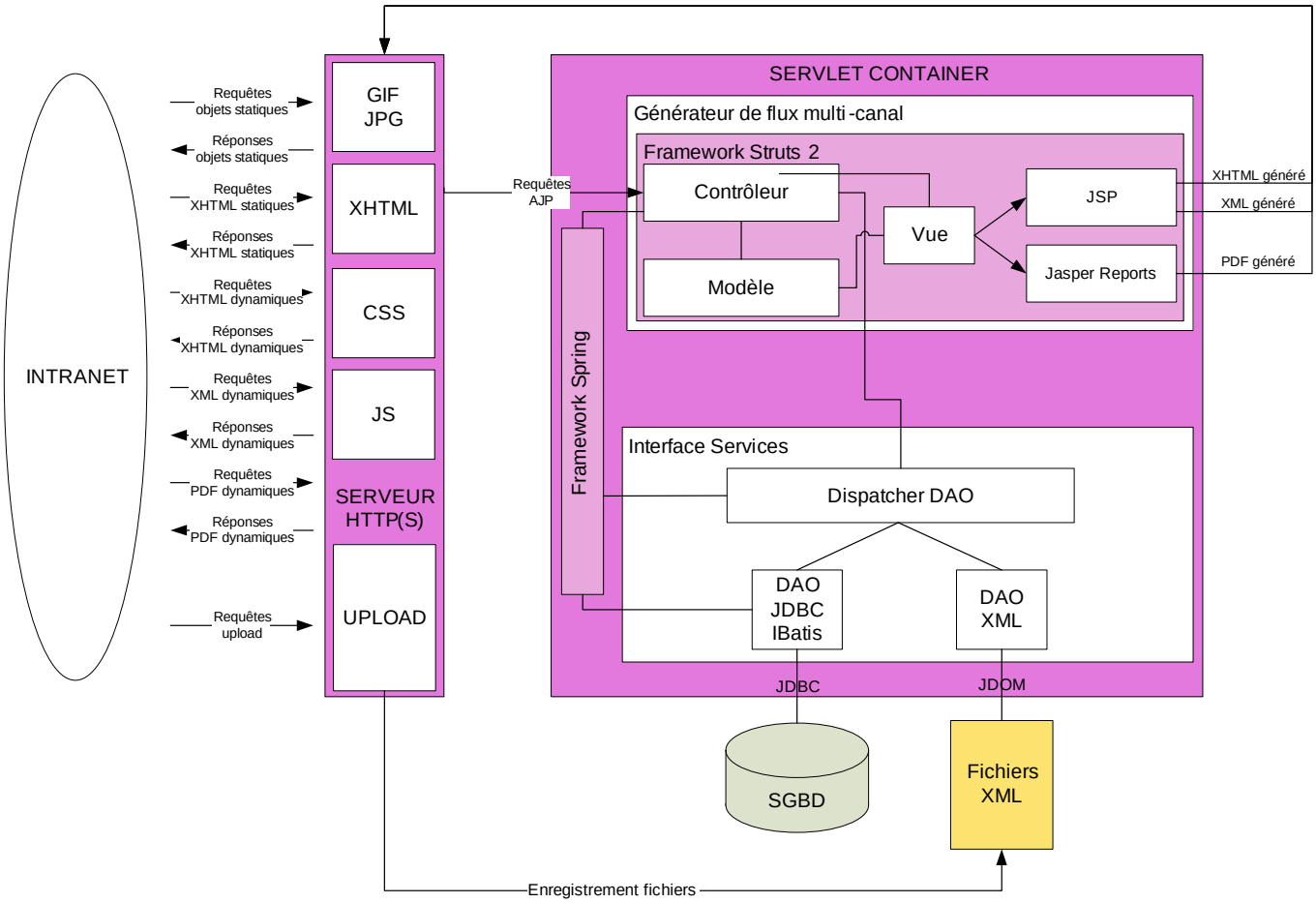

Figure 4 : Architecture globale du serveur ACube (Lise 3.x)

Cette figure illustre les principales briques **Open Source** de l'architecture **multi-niveaux** ACube (Lise 3.x), soient :

- Apache pour le serveur Web.
- Tomcat pour le « Servlet Container » et le moteur JSP.
- Connecteur AJP entre Apache Web Server et Tomcat.
- JRE 1.5.0 comme moteur d'exécution Java.
- Spring pour l'injection de dépendance entre les couches Web, Services et DAO
- Struts 2.x pour la navigation et la génération de flux XML générés
- Jasper Reports pour la génération de PDF
- IBatis pour l'implémentation des DAO
- SQL Server, Oracle ou MySql pour la base de données relationnelle.

4 FRAMEWORK DE DEVELOPPEMENT ACUBE

4.1 FRAMEWORK CLIENT RICHE W3C FRED

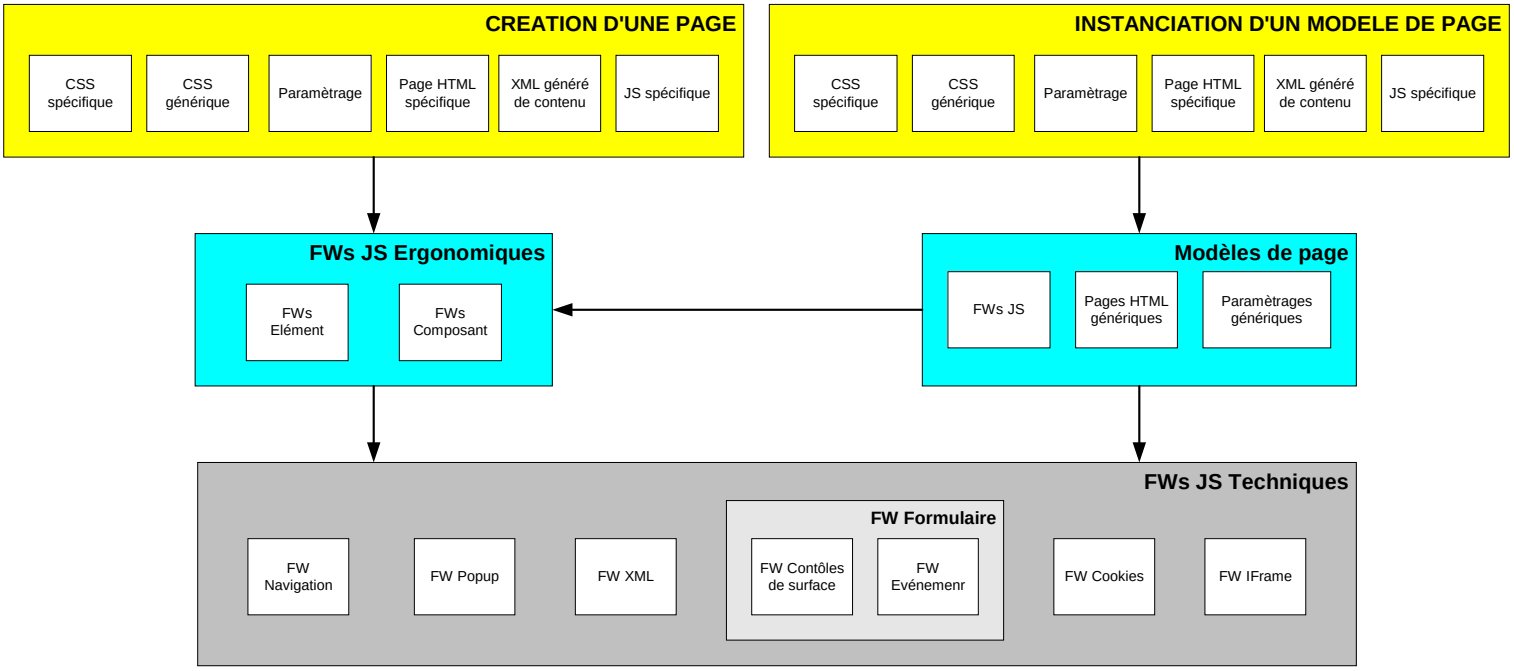

Figure 5 : Découpage du framework ergonomique JavaScript client riche

Cette figure présente une description structurelle du framework ergonomique à implémenter.

Ce découpage du framework ergonomique en plusieurs frameworks JavaScript a pour objectif de répondre à deux besoins de développement :

- **Instancier une nouvelle page à l'aide d'un modèle de page.**

  Cette instanciation consiste alors juste à effectuer un paramétrage spécifique au cas d'utilisation métier et à ajouter les codes (X)HTML (positionnement des éléments constitutifs dans un tableau…) et JavaScript (contrôles de surface…) spécifiques. Le code HTML spécifique est ajouté au code HTML fourni par le modèle de page.

- **Développer une nouvelle page ne répondant pas à un modèle de page existant.**

  Ce développement peut bénéficier des frameworks ergonomiques déjà présents pour faciliter sa mise en œuvre. Dans ce cas, le développement nécessaire reprend l'ensemble des contraintes décrites lors d'une instanciation mais avec la nécessité de développer l'ensemble du code HTML d'une part, les éléments et composants spécifiques à ce cas d'utilisation d'autre part. Avant d'effectuer ce développement, il sera intéressant d'étudier si ce développement peut être mutualisé soit pour ses éléments, soit pour ses composants spécifiques, voire même de réaliser l'objet d'un modèle de page.

C'est dans cette optique que ce découpage suit le principe de « poupées russes » pour découpler et surcharger les frameworks répondant aux problématiques suivantes :

- **Techniques :**
  - « Framework Navigation » mutualisant l'ensemble du code nécessaire à la navigation en prenant en compte les problématiques de présence de frames et/ou d'iFrames. Ce framework gère aussi l'intégration ou non de la navigation dans l'historique du navigateur.
  - « Framework Popup » pour la gestion des fenêtres surgissantes avec spécification du rendu de la fenêtre, de la gestion de la demande d'impression de son contenu, du caractère d'unicité de cette fenêtre…
  - « Framework XML » pour la mutualisation de la gestion des flux XML vis à vis des appels, des messages d'attente, de la gestion des erreurs et du TimeOut, de l'appel de la fonction liée à la réception du flux XML, de la sauvegarde des flux XML sur disque… Ce framework fournit aussi une boîte à outils d'interrogation et de manipulation avancée de ces flux XML (Tri, test de valeur optionnelle ou de sous-arbre vide, test de la typologie du nœud XML avant interrogation ou manipulation…)
  - « Framework Formulaire » pour gérer les contrôles de surface sur la saisie du client, de valider le formulaire par la touche ENTER lors de la saisie et d'empêcher les validations non désirées tant que la première validation n'a pas été réalisée.
  - « Framework Cookies » pour la création, interrogation, manipulation et suppression des cookies session.
  - « Framework IFrame » pour la gestion d'IFrame dans une page en mutualisant l'interrogation et la mise à jour de son contenu suivant le type d'IFrame désiré (taille fixe avec scroll-bar ou taille en fonction du contenu).
- **Ergonomiques :** génération et gestion d'éléments et composants liés à la charte graphique et aux besoins des applications ciblées par l'entreprise. Ce framework permet de pouvoir répondre aux deux besoins de développement et de faciliter la maintenance future par son effort de mutualisation.
- **Modèles de page** : surcharge du framework ergonomique pour gérer un ensemble d'éléments ou de composants dans le cadre d'un modèle de page.
- **Métier** : ne faisant pas partie du framework ergonomique mais d'un cas d'utilisation donné sous la forme de fonctions ou de bibliothèques ou de frameworks JS spécifiques.

Les frameworks client riche W3C sont entièrement détaillés dans des documentations dédiées (JSDoc).

4.2 FRAMEWORK SERVEUR J2EE LISE

4.2.1 FRAMEWORK LISE V 2.X

4.2.1.1 STRUTS

Struts, framework de classes Java créé en open source dans le cadre de l'ASF (Apache Software Foundation), implémente complètement le modèle MVC2 en proposant un mécanisme d'automatisation des relations entre servlets, JSP et objets métiers. Cette intégration se fait par le biais des classes ActionForms.

La manipulation des classes ActionForms au sein des JSP est grandement facilitée par l'emploi des taglibs essentiellement dédiées à cet usage. Struts fournit donc un peu plus d'une cinquantaine de tags personnalisés regroupés au sein de quatre librairies (taglibs) :

- La librairie des tags bean pour la manipulation pure de JavaBeans,

- La librairie des tags HTML pour la manipulation des formulaires HTML,
- La librairie des tags logiques pour la mise en place de traitements conditionnels et/ou itératifs,
- La libraire des tags template qui propose un mécanisme de gestion de modèles de JSP (les templates).

En synthèse, Struts propose de construire les applications Java autour d'une organisation de l'Interface Homme Machine en implémentant le modèle MVC2.

Cette organisation apporte un respect net de la séparation entre présentation (JSP) et contrôle du dialogue (Actions) en excluant autant que faire se peut tout code Java des JSP (utilisation des taglibs).

Par ailleurs, la centralisation des accès à l'application via le contrôleur permet un contrôle fin et personnalisé des traitements et offre une grande marge de manœuvre pour la gestion des profils ou des rôles utilisateurs.

4.2.1.2 STRUTSCX

StrutsCX, framework développé en Open Source (SourceForge.net) combine à la fois les avantages de Struts avec ceux d'un moteur XSLT. Il s'installe à côté du framework Struts. Il remplace la JSP de la Vue par l'utilisation d'un moteur XSLT tout en gardant les parties Contrôleur et Modèle de Struts. Il implémente entièrement le modèle MVC2X et associe à la simplicité de Struts l'ouverture vers de multiples formats (XML, PDF, SVG, XLS…).

De plus, la gestion d'erreur de strutsCX est compatible avec l'ActionErrors de Struts pour pouvoir remonter au contrôleur Struts tout problème rencontré lors de la génération de la présentation pour une action donnée.

La transformation XML s'effectue en transformant un objet par un flux connu sous le nom de « serialization ». Les objets reflétant les objets métier sont déclarés en objet « serializable ». Ainsi, la Vue ne sollicite pas une page JSP mais une servlet qui organise la construction d'un objet XML prêt pour être transformé par le biais d'une feuille de style XSL. Il est possible de vérifier la cohérence du fichier XML à l'aide de DTD.

Ce modèle 2X permet donc :

- **d'une part de concilier** :
  - Une ergonomie complexe basée sur les principes du client riche car le moteur XSLT est capable de générer des flux XML à partir du XML-Output-Document auto-généré à partir des informations contenues dans la couche « modèle » du MVC.
  - Formaliser un contenu informatif en générant des flux XML dont le format est validé par la transformation XSLT.
  - La production simple de fichier d'édition ou de statistiques au format PDF, SVG, CVS…
- **d'autre part de s'affranchir des contraintes** suivantes liées au JSP facilitant la programmation Web dynamique avec les Tag Librairies mais possédant un certain nombre de limitation :
  - Orientation vers une ergonomie simple basée sur une navigation par page ou validation d'un formulaire.
  - La syntaxe JSP est moins souple que le XML.
  - Les programmeurs peuvent mettre de la logique applicative dans leur JSP au lieu de se cantonner à une logique de présentation.
  - JSP n'est pas entièrement compatible avec le XML et est incapable de garantir à 100% que les documents XML ou XHTML générés sont correctement formatés.
  - Impossibilité de séparer le style du rendu.
  - Souplesse ergonomique et persistance de données difficiles car seule les poursuites d'action et les inclusions JSP sont possibles.
  - Recompilation des JSP nécessaires pour chaque changement de la présentation demandant un temps non négligeable lors de modifications de rendu visuel.

Pour toutes ces contraintes, il est apparu qu'il était plus intéressant de remplacer la technologie JSP par une transformation XSLT en ne modifiant rien à Struts et aux autres avantages acquis (séparation Contrôleur/Modèle/Vue).

On obtient donc le modèle MVC2X suivant :

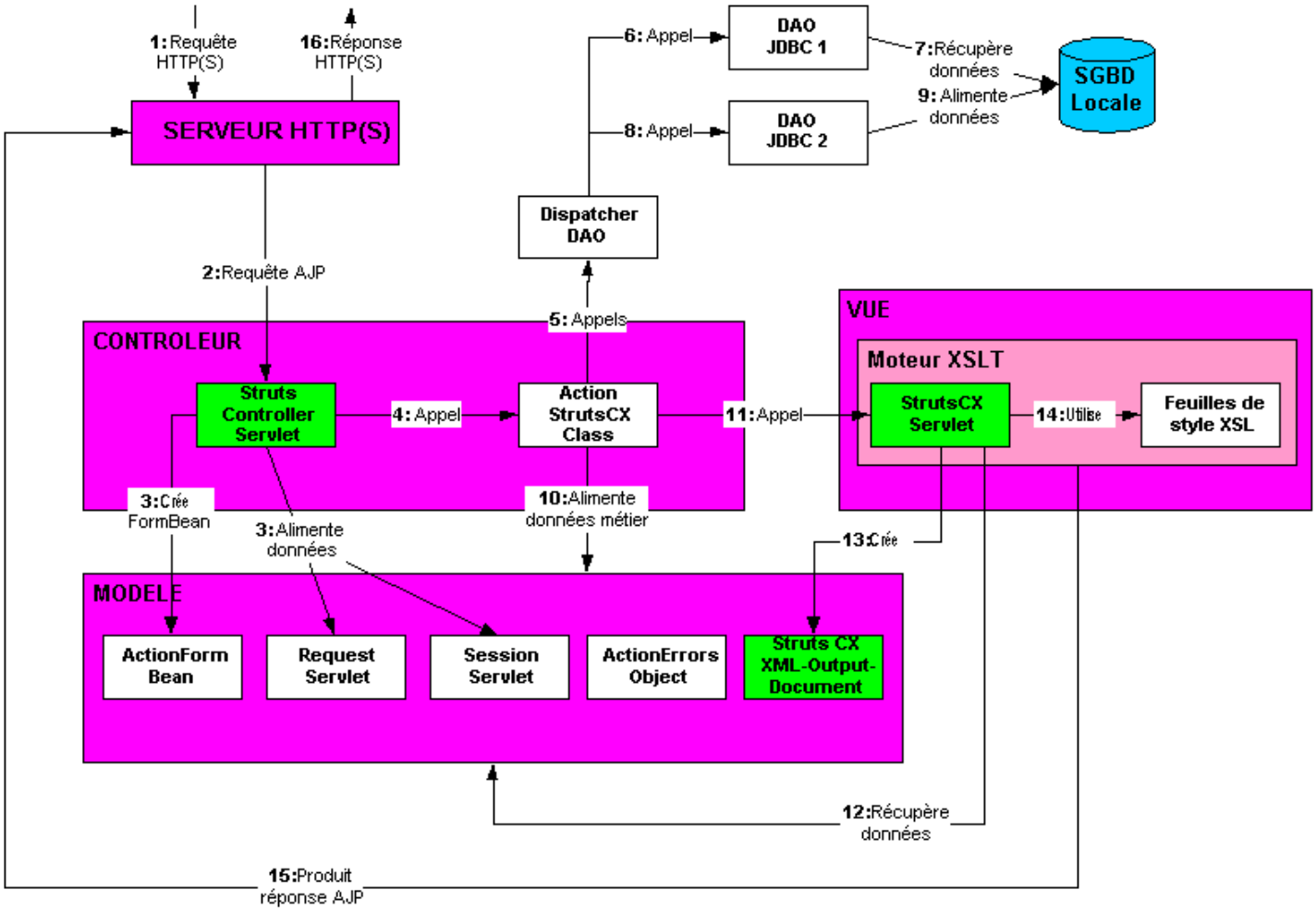

Figure 6 : Framework StrutsCX pour un modèle MVC2X

A partir du document XML-Output ajouté à la couche « Modèle » par le framework StrutsCX, la couche « Vue » est capable de spécifier le format de sortie de la génération de flux dynamiques en effectuant un mapping sur une feuille de style XSL en fonction de l'action ou de la requête effectuée par l'utilisateur. Ce mapping précise au moteur XSLT Xalan la feuille de style associée pour générer du XML formaté et validé. Il est aussi possible de demander que le document XML-Output soit directement renvoyé en initialisant le paramètre « debugxml=true » soit dans la requête client ou soit dans l'Action Class Struts CX.

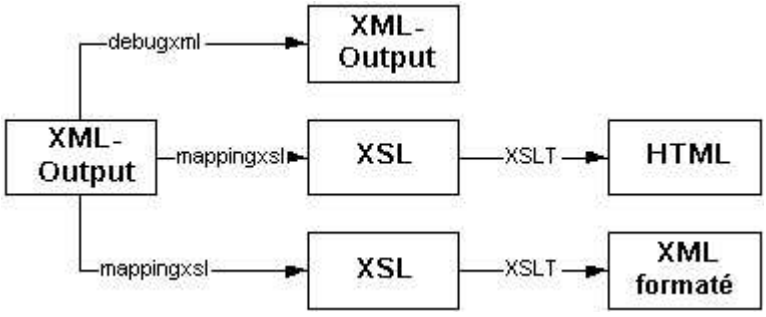

Figure 7 : Framework StrutsCX pour générer du XML formaté ou de l'HTML

Il est possible de spécifier un format de sortie PDF grâce à l'intégration du moteur XSLT Apache FOP. Ce moteur permet de générer des objets formatés à l'aide de tags FO implémentés dans la feuille de style (XSL-FO). Cette solution permet ainsi de capitaliser le code métier et de rapatriement de données pour fournir une sortie sous forme de version imprimable en PDF.

Figure 8 : Framework StrutsCX pour générer du PDF

4.2.1.3 JDBCWrapper

Les différents composants fonctionnels feront l'objet d'une analyse lors de la phase de spécifications techniques détaillées. Les classes résultantes, appelées objets métier, doivent être indépendantes de toute interface utilisateur ou mécanisme de persistance. Elles offrent une abstraction qui facilite la manipulation des données métiers.

Cette couche d'objets métier permet de construire des objets à partir des tables relationnelles, et de mettre à jour ces tables lorsque les objets sont créés, modifiés ou supprimés.

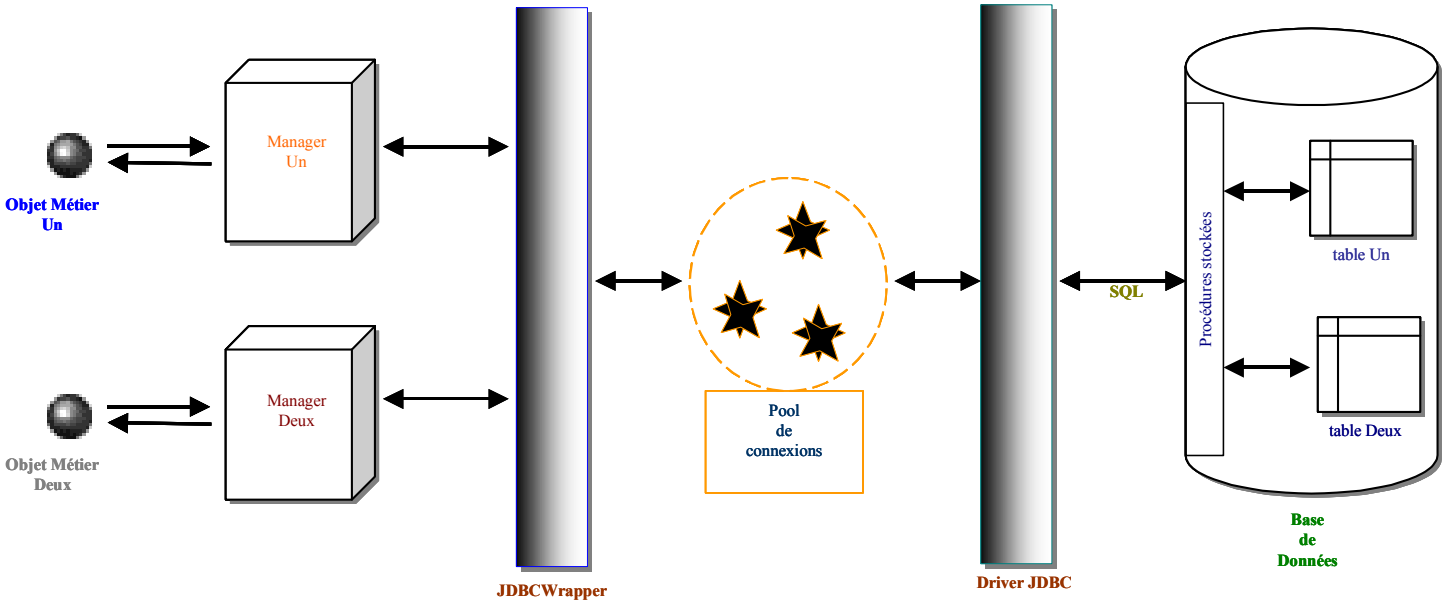

Figure 9 : Framework JDBCWrapper

En ce qui concerne la persistance des objets, la solution est de mettre en œuvre des classes dites « **Manager** » qui encapsulent les requêtes d'accès aux tables de la base de données. Techniquement, ces "Managers" s'appuient sur un framework « **JDBCWrapper** » qui masque la gestion des connexions à la base de données. Pour des raisons de performance, les connexions seront issues d'un pool qui permet une meilleure optimisation des ressources. La mise en place du **pool de connexion** s'effectue à partir de DBCP (Projet commons d'Apache).

Finalement chacune des requêtes d'accès à la base est relayée par un **driver JDBC** compatible avec la base de données cible (driver JDBC de jTDS pour SQL Server par exemple). Les requêtes permettent d'exécuter des procédures stockées de la base de données ou d'encapsuler directement le résultat d'une requête SQL.

Ce framework permet aussi de **compartimenter le langage SQL du code Java** sous la forme de fichier monté en mémoire pour pouvoir ainsi isoler l'ensemble des requêtes SQL utiles à un applicatif et séparer les deux logiques de programmation dans un souci de faciliter la maintenance future et de mutualiser les requêtes.

Via le framework « JDBCWrapper », il est aussi possible d'encapsuler dans une même transaction via un « Manager » d'encapsulation plusieurs objets métiers et leurs « Managers » associés pour gérer ainsi un « commit global » sur la transaction ou un « roll back global » en cas d'une erreur survenue sur un des « Managers » d'un des objets métiers.

4.2.2 FRAMEWORK LISE V 3.X

4.2.2.1 SPRING

Le framework Spring a été développé dans l'esprit d'offrir une infrastructure permettant le développement d'application Java d'entreprise, en s'affranchissant des lourdeurs de la norme J2EE, et notamment des EJB. Pour cela, Spring propose une conteneur léger de Java Beans, gérant la cycle de vie des objets, et reposant sur le pattern de l'injection de dépendance.

Habituellement, pour initialiser un objet qui dépend d'un autre objet, deux stratégies pouvaient être envisagées : l'utilisation de l'opérateur new, ou la mise en œuvre d'une factory, souvent codée par le développeur.

Spring offre une factory générique paramétrable via des fichiers XML décrivant les dépendances entre les classes de l'application. Une fois les dépendances décrites, Spring gère l'instanciation des objets et leur injection dans les objets dont ils dépendent.

Cette rupture de dépendances dures entre les objets offrent plusieurs avantages :

- Modularité des composants développés
- Testabilité accrue

Dans le domaine des tests, Spring offre également un plus value importante en offrant une intégration poussée avec JUnit pour permettre de tester rapidement, et dans l'IDE, l'intégration des différents composants.

4.2.2.2 STRUTS 2

Struts 2 est le résultat de la fusion de Struts (cf §4.2.1.1) et du framework WebWork, mais proposant une rupture complète avec Struts 1.x. Struts 2 implémente également le modèle MVC2, en offrant une architecture beaucoup plus souple que Struts 1.x, notamment par l'intégration des éléments suivants :

- Support de l'injection de dépendances au niveau des actions (via Spring)
- Utilisation d'OGNL pour le mapping des valeurs de formulaires, et possibilité d'utiliser n'importe quelle classe comme une action, n'obligeant plus d'étendre les classes Action et ActionForm et rendant possible l'utilisation de simple POJO pour coder les actions
- Mécanisme d'intercepteurs pré et post traitement permettant la factorisation de traitements communs

Ces évolutions, tout en gardant tous les avantages de Struts 1.x (séparation nette des couches, centralisation des accès à l'application), offrent en particulier aux applications ACube une testabilité, sous forme de tests unitaires des classes actions, beaucoup plus importante, aidée par l'indépendance de Struts 2 de l'API Servlet.

4.2.2.3 iBatis

iBatis, projet de l'ASF (Apache Software Foundation), est un framework de mapping objet-relationnel, permettant la mise en place d'une correspondance entre des requête SQL et des instances d'objets, en se basant sur des fichiers de description au format XML.

Associé au framework Spring (en particulier son module Spring DAO), et par l'intermédiaire de son outil iBator, iBatis permet la génération automatique des VO et des DAO (cf §2.3.1) en se basant sur le schéma de la base de données, en proposant un mapping automatique des types JDBC sur les types de base du langage Java. L'association avec Spring permet également à IBatis de s'appuyer sur le support poussé des transactions offert par celui-ci.

Utilisant JDBC, iBatis est compatible avec toutes les bases de données disposant d'un driver JDBC.

4.2.3 Log4J

Conformément au principe de traçabilité décrit au §2.4, l'utilisation d'un **Framework de gestion des logs (Log4J)** couplée à une récupération systématique de toutes les exceptions Java permettent d'obtenir un code fiable et facilitent la maintenance future de l'application.

Plusieurs « loggers » orientés ACube permettent ainsi de garantir l'implémentation de la méthodologie retenue dans le cadre de la filière de développement ACube.

4.2.4 jBPM

jBPM, projet de la communauté JBoss, est un moteur de workflow, c'est-à-dire un middleware permettant la modélisation et l'exécution d'instance de processus métier (workflow), et s'interfaçant avec un autre logiciel, ici une application ACube.

jBPM permet la modélisation de processus métier sous la forme de graphs d'état orientés, et utilise son la définition de ces graphs son propre langage de définition : jPDL, offrant comme avantage principal une intégration très propre avec Java.

La description de processus est stockée dans des fichiers au format XML, et jBPM offre un outillage puissant afin de pouvoir réaliser graphiquement cette définition, sous la forme d'un éditeur WYSIWYG.

jBPM s'intègre particulièrement bien avec le serveur d'application JBoss SA, offrant un conteneur aux EJBs de jBPM pour un accès distant à ses APIs.

4.2.5 Drools

Drools (aussi appelé JBoss Rules), projet de la communauté JBoss, est un moteur de règles, c'est-à-dire un middleware permettant la définition et l'exécution des règles métiers d'une entreprise, définies de manière déclarative en entrée du moteur.

S'appuyant sur un modèle métier permettant de définir une base de faits, et sur une base de connaissances (la définition des règles métiers de l'entreprise) (offrant une séparation nette entre la logique et les données), Drools permet la création d'un système expert.

Drools est un moteur à inférence, c'est-à-dire déclenché par une action de l'utilisateur. Un plugin pour l'IDE Eclipse est proposé, facilitant l'implémentation et le test unitaire des règles.